TRABAJO PRÁCTICO ESPECIAL 1

Segundo Cuatrimestre de 2023
72.42 - Programación de Objetos Distribuidos

Grupo 6
Mattiussi, Agustin 61361
Ortu, Agustina Sol 61548
Sasso, Julian Martin 61535
Vasquez Currie, Malena 60072

Índice

1. Decisiones de diseño e implementación de los servicios

1.1. Clases Propias

Contamos con las clases Ride y ParkPass las cuales refieren a las atracciones y los pases al parque respectivamente. Los pases al parque guardan información del visitante que lo compro, que tipo de pase es y para que día. Luego las atracciones cuentan con su información como nombre, tiempo de apertura, cierre y la cantidad de minutos de los slots para esa atracción (clase propia RideTime), un mapa que indica la capacidad de cada slot para cada día y un mapa que indica para cada día, en cada horario, la lista de reservas.

1.2. Slots

Para las reservas se tomo la decisión de que al aplicar los slots, si hay tiempo .extra", en este no se pueden tomar reservas. Por ejemplo, si mi atracción abre de 09:00hs a 10:00hs y los slots son de 40 minutos, tendremos un único slot de 09:00hs a 09:40hs, los 20 minutos restantes entre las 09:40hs y 10:00hs no podrán recibir reservas dado que los minutos de slot son mayores a 20 (la atracción cierra).

1.3. ParkLocalTime

En cuanto a los horarios del parque, se decidió implementar la clase *ParkLocalTime* como *wrapper* de *LocalTime* (dado que esta última es una clase *final* y no puede ser extendida). La misma, implementa métodos para facilitar el manejo de horarios en formato *HH:mm*.

1.4. Stubs

En primer lugar se planteó la idea de hacer uso de BlockingStub para los clientes, pero luego tomamos la decisión final de hacerlo con FutureStub dado que nos pareció que era de mayor importancia tener en cuenta la asincronía y un mayor control de concurrencia y escalabilidad, frente a la simplicidad de código y sincronía del blocking.

2. Criterios aplicados para el trabajo concurrente

2.1. Colecciones Concurrentes

Con el fin de mejorar el trabajo concurrente al momento de guardar información, se optó por utilizar colecciones concurrentes en lugar de las tradicionales. Dentro de estas se encuentran *ConcurrentHashMap* y *ConcurrentSkipListSet* que, al ser thread-safe, permiten el acceso de multiples threads en un mismo instante sin que haya problemas de sincronización.

Otra ventaja del *ConcurrentHashMap* es que realiza un locking muy detallado lo cual lo hace más eficiente para operaciones concurrentes. Por su parte, *ConcurrentSkipListSet* implementa *SortedSet*, lo que facilita la búsqueda de *Reservation* manteniéndolas ordenadas (nótese que *Reservation* implementa *Comparable*)

2.2. AtomicInteger

Se optó por utilizar AtomicInteger en lugar de *int* con *synchronized* ya que, al incluir los métodos *incrementAndGet(), decrementAndGet()*, evita problemas de concurrencia a la hora de actualizar valores. También encapsula la sincronización de forma que asegura un mejor manejo de threads.

En particular, esta clase se utilizó para el control de los espacios disponibles para cada atracción, asegurándonos de evitar el caso donde dos hilos intentan modificar esta variable al mismo tiempo. Adicionalmente, también se usó para el conteo de las atracciones agregadas y fallidas en el cliente de administración, ya que el mismo espera varias respuestas que pueden llegar en simultáneo.

2.3. Synchronized

Para evitar que dos clientes del servicio de administración intenten configurar la cantidad de espacios disponibles para cada atracción en simultáneo (o peor aún, en el medio de la ejecución de alguno de ellos), se decidió la implementación de un bloque *synchronized* sobre el mapa *slotCapacityByDay*. De esta manera, el segundo hilo en llamar a la función de asignación de espacios, deberá esperar a que el primero libere el mapa para luego fallar, tras comprobar que ya se configuró una capacidad para el mismo día.

2.4. Lock

En la clase *Reservation*, existe un *booleano* "*shouldNotify*"que indica si existe un cliente suscripto a las notificaciones para la instancia de reserva correspondiente. Para suscribir un usuario a notificaciones, se verifica y actualiza *shouldNotify* y luego se guarda en la instancia un *notifiationObserver* recibido como parámetro. De forma opuesta, para la desuscripción se elimina el puntero a este *observer* y se actualiza *shouldNotify*. Con el objetivo de atomizar estos métodos, que pueden llevar a condiciones de carrera, se decidió implementar el *Lock* "*shouldNotifyLock*".

De forma similar, para la verificación y posterior modificación de la cantidad de espacios disponibles en una atracción (en caso de una nueva reserva o una cancelación), se implementó el *Lock* "*slotCapacityLock*".

Por útlimo, notamos que existe una importante condición de carrera cuando se consulta si ya se cargó la capacidad de espacios disponibles por día en una atracción. La misma ocurre debido a que tras configurar este valor, deben reubicarse las reservas sobrantes de cada horario. ¿Qué ocurría si ocurre un cambio de contexto, por ejemplo, para realizar una reserva, luego de que se definan los espacios pero antes de que se reubique? Para solucionar esta situación, dado que los espacios sólo dependen del día, se implementó un *ReentrantLock* para cada día del año (dentro de una misma ride). El mismo, permite la paralelización de la carga y reubicación de espacios para distintos días, a la vez que detiene las reservas de un día hasta que termine la reubicación en el mismo.

2.5. Latch

En los clientes, se utilizan instancias de *Latch* para evitar que los mismos finalicen su ejecución hasta no haber recibido la respuesta (o respuestas) del servidor.

3. Potenciales puntos de mejora y/o expansión

3.1. Persistencia

Los archivos *BicycleServiceRepository.java* y *ParkPassRepository.java* funcionan como Stores o repositorios para los datos lo cual, si bien cumple con lo pedido, solo retienen la información durante una misma sesión. Como mejora a futuro, se podría implementar una base de datos real que permita tener persistencia.

3.2. Comunicación

Un planteo que sería interesante para implementar es en el caso de agregar atracciones y agregar pases. Esto se podría materializar usando streaming, de forma que se envien todas las atracciones o pases (según corresponda) como stream al servidor. Se esperaría que devuelva el conteo de cuántas se agregaron y cuántas no.

3.3. Gestión de Reservas

Sería ideal tener dos mapas de reservas, uno para reservas confirmadas y otro para las reservas pendientes. De esta manera evitariamos recorridas innecesarias para el conteo de estados y se podía acceder de forma mas directa al tamaño de esos mapas según corresponda.. Esto podría significar una mejora en la claridad, organización y eficiencia en la gestión de reservas.

Otra posibilidad que ayudaría a mejorar la eficiencia sería tener un mapa <VisitorId, <Day, Reservation>>ya que permitiría encontrar de forma directa las reservas del usuario. Esto es más óptimo en comparación con recorrer todo el mapa de reservas en un dia dado.